Automated generation of dependency models for service management

Christian Ensel[*]

Munich Network Management Team
University of Munich, Dept. of CS
Oettingenstr. 67
D-80538 Munich, Germany
Phone: +49-89-2178-2171, Fax: -2262
E-Mail: ensel@informatik.uni-muenchen.de

Abstract

Various service management tasks depend on knowledge about objects they are applied to. Understanding dependencies between these objects is of special interest. Such dependency models are neither generally available nor easy to create. Creating them manually is expensive in regard to both time and money. In addition, they must be brought up to date, reflecting the frequent changes in the managed environment.

After motivating use of dependency models for various service management tasks, this article presents a new approach on the modelling itself. It is shown that the proposed modelling process enables automated generation of such models. To add flexibility and to guarantee applicability in many environments a special technique based on neural technologies is used. This avoids or at least reduces the disadvantages of a purely manual creation process.

[*]The author's work is sponsored by the "Bundesministerium für Bildung und Forschung" (BMBF) project "LEONET"

1 Introduction

Service management is becoming more and more important within the area of IT-management. Its significance is stressed by the nowadays commonly used expression of the *paradigm shift* towards service management — expressing that IT-management is no longer focused on components enabling the services, but driven by top down requirements. These have their origins in demands of service customers, contracts specifying "quality of service" (service level agreements) and company policies.

The problems of service management, such as specifying these requirements, breaking them down to components, etc. have not been solved yet. However, with the subject becoming more and more important, also more work on this subject is being carried out.

One special difficulty arises from the fact that services cannot be considered isolated tasks. They tightly depend on other (sub-)services and — on the lower level — on operating systems, physical components and communication infrastructure. Obviously, several tasks for service management benefit from — or are even impossible without — the knowledge about inter–service dependencies. Such tasks are described in section 2.

Descriptions of such dependencies of services are commonly called service dependency models. Section 3 gives more exact definitions of several distinct types of models.

In section 4 this article describes what existing problems, mainly concerning the model's manual creation. Later, section 5 presents a process enabling the automated creation of models along with discussions of the resulting benefits. As the project is still in its beginnings, a prototype cannot be presented yet; nevertheless, an analysis of some aspects of implementation follow in section 6.

2 Existing Applications of Dependency Models

Several research projects (e.g. [12], [2]) already investigated the advantages of models of dependencies between services. A common result of these examinations is that — assuming models do already exist — great benefits can be achieved for management tasks.

One application is the so-called *root cause analysis*. It helps to find a common (root) cause of problems or faults detected at different places within an environment. It may be applied to network components reporting error conditions as well as on services where, e.g. the service users detects the problem. The reason for the actual need of such root cause analysis is that error conditions or problem reports brought to the administrators or management systems, are just description of symptoms. To be able to derive their causes, further knowledge about the dependencies among them is necessary. [6], [5] and [9] explain this subject in detail.

Similar dependency models are needed when *determining availability requirements* on services from superior ones (looking from a top down perspective) respectively for the *calculation of service availability* from the availability of underlying services (bottom up), as described in [10].

The management on the lower OSI layers also benefits from such models. E.g., [11] applies reasoning on models for network performance management.

The knowledge of dependencies between services may further be useful for the *prediction of impacts* on other services due to management operations. This is of particular interest in the typical 'repair'-scenario, where a service implementation has to be shut down temporarily: it might be essential to know the effects on other services beforehand.

3 Types of Models

To enable a common understanding of the meaning of dependency models in the explanations following later, this section distinguishes and explains several model types.

Generally speaking, *service dependency models* can be considered graphs consisting of nodes which represent the managed objects — in this case the services or service realizations — and of edges, standing for functional inter-service dependencies.

In the simplest case the graph is not directed; then edges only represent generic relationships, but it is not possible to express the direction of the dependencies. This must be regarded a serious deficiency for lots of applications, but might be sufficient for special tasks.

Usually, directed graphs are used. Besides the direction it is in some cases useful to attach further management relevant attributes to them.

With these it is possible:

- to form groups which, e.g., to express that a dependency cannot occur without the others,
- to express that some dependencies must occur in a certain order, or
- to attach values of strength or likelihood.

One can already imagine that dependency models used for different purposes may differ in various ways, e.g. according to their level of abstraction, the degree of detail and of course the types of services and components. Examples are:

- hosts with communication dependencies
- high level services, disregarding distributed service implementations and
- services including several levels of subservices.

Basically, for the models there are always two opposite levels of abstraction:

1. abstract service models and

2. models of real world services (applications, components, etc.).

The first type is used to model purely abstract services with their dependencies. It is independent of any concrete runtime environment. Figure 1 shows an example.

Advantages of abstract representations are that they are comparably small, compared to the second type, generic in nature and reusable in various environments.

Models of the second type provide a representation of a particular, real environment — it models the dependencies of the service implementations as they exist at runtime, like a program or piece of hardware communicating with others, or depending on underlying software/hardware components. Thus, the infrastructure the services are implemented in influence the structure of the model.

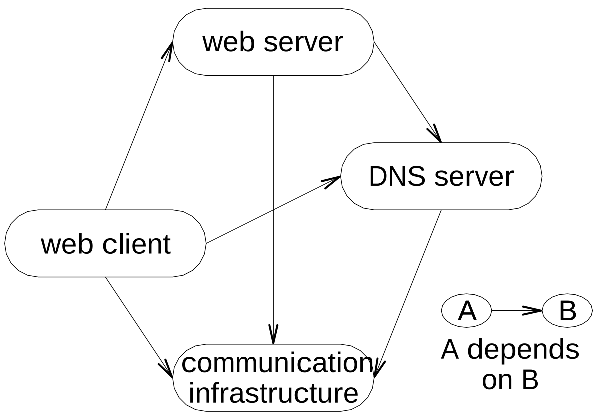

Figure 1: Typical Abstract Service Dependency Graph

Sometimes these models are simply considered as instantiated examples of the abstract graph. However, while this is true for the nodes (services), this may not be true for the edges (dependencies). It may, e.g., not be the case due to special circumstances, where the abstract model is (purposely) generalising too much, thus hiding differences in distinct service implementations.

Figure 2 shows a scenario, where one of the web clients does not depend on the DNS server. This might be the case, because only very special web pages are viewed from that particular browser, for which only direct IP addresses are needed, or simply because the host names are stored in a static configuration file. Thus, the dependency depicted by the broken line would be part of a model instantiated from the abstract one (figure 1), but is not part of the real world model. For simplicity reasons, the modelling of the communication infrastructure is left aside.

Thus, the

3. instantiated dependency model

is a third type of models.

Compared to the abstract models, a possi-

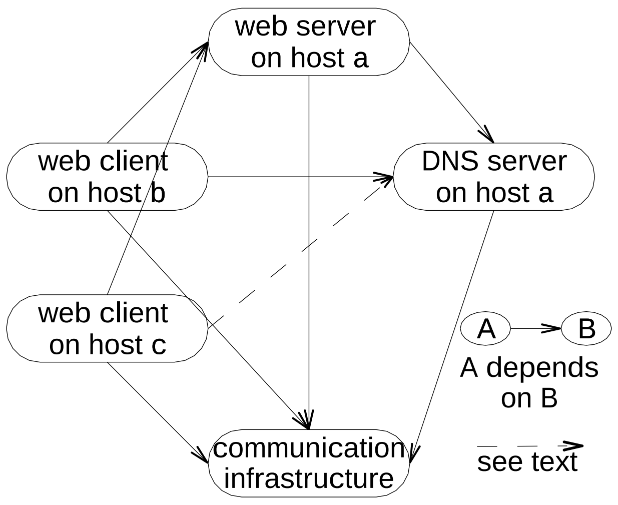

Figure 2: Real World Service Dependency Graph

bly huge number of objects and dependencies might be needed for the real world and the instantiated models. The real world graph has less or exactly the same number of dependencies than the instantiated one, otherwise there would be major faults in the abstract model.

In the differences between these two graphs lies useful information, because they are, in other words, exceptions from the generic case.

Models, as usually described in other works, are either abstract or instantiated models. The benefits of models that directly describe real world dependencies are usually left out of consideration, because the automatic creation is not yet possible in an adequate way.

Between these two extremes of abstract and real world models lie the

4. reduced real world model.

Just like the real world models, they provide a view on the real services and resources, but — to reduce complexity — they make one step towards abstraction by merging some instances into one node.

This may, happen to hide the existence of redundant services, or with services/applications of the same type used exclusively. The latter case is useful, e.g. in a scenario where a worker may use one of two input terminals, according to his location at the working environment (e.g. the side of the assembly line), but not both at the same time.

4 Problems of Manual Model Creation

Section 2 already mentioned that most papers on the benefits of dependency models assume the existence of such models. Others, like [2] present means of describing models suitable for IT-management, but also do not provide models as such.

Despite of the positive results of such investigations, dependency models are hardly used at a larger scale in real environments. This is due to several reasons:

Up to now, they have to be created by hand; this can be a very *time consuming* task, especially in complex environments with a large number of services. The problem gets worse if different applications of the dependency models are taken into account which need models at different levels of detail. Either, one finds some kind of compromise or has to create several distinct graphs.

From a purely top down perspective, the 'best' models are based on generic services; reusable in various environments. However, to guarantee a wide acceptance, it is finally up to international standardisation organisations to define generic descriptions of services and their dependencies — usually taking a very long time.

Examples for generic definitions can be found in the Common Information Model (CIM, [3], [4] respectively), defining classes for services and other managed objects including several types of dependencies.

Later, during the utilisation of models, it might be necessary to change some of them, adapt them to new conditions, etc. This is especially a problem, because services often cannot be modelled on a very abstract level, but — as programs available on the market do not always exactly match the exact requirements — the models must consider the properties of the real world service realizations. Even simple software updates might then lead to changes in the models. Over the years, this again sums up to a time consuming task.

Preventing the design of widely used models is the *lack of an established common model description format* on the market. This makes the realization of management applications more difficult and is a real handicap for reusing the same models for different purposes. Actually, this problem should, e.g. be addressed by the information model of management platforms — but nowadays these are not yet powerful enough to handle the kinds of dependency descriptions needed.

The above reasons finally lead to the fact that *companies do not deliver models along with their products.* This is a further handicap for their customers. The only dependencies usually described are vaguely expressed prerequisites like “needs HP-UX10 and 200MB of disk space”.

There are further reasons, why companies do not deliver more exact dependency descriptions: On the one hand due to confidentiality of the information, and on the other hand due to strategic reasons which are enforced to help strengthening the companies market position.

The big efforts that have to be taken to generate the models also do not allow to carry out modelling regularly. This prevents their application for special management tasks, as described in the end of section 5.

All these disadvantages are reasons why dependency models are not widely used. One attempt to overcome — at least the most significant — disadvantages is to automate the creation process of models as much as possible. This would help to reduce the time to create, as well as to maintain the models, to an acceptable extent and thus help to diminish the problems coming from the companies (still) not delivering models with their products.

5 Overall Modelling Process

Before an explanation of the modelling itself, this section describes the overall modelling process it is embedded in.

Usually, the manual creation of abstract models is one of the first steps. Afterwards, the instantiated model is derived adding knowledge about the environment.

However, to enable the automated creation of such models, it is necessary to chose a different approach: The abstract models have to be created from real world dependency models, because it is possible to directly construct them from noticeable interactions of or between services implementations.

The input for the process is a list of (abstract) services and component types for which dependencies should be modelled, and a matching list of service implementations and real components. They will finally become the nodes of the abstract, respectively the real world dependency graph. The output is one abstract and one real world model, containing all dependencies detected.

Figure 3 depicts the necessary steps. The left part of the figure represents the abstract view on services, whereas the right side deals with their real world realizations:

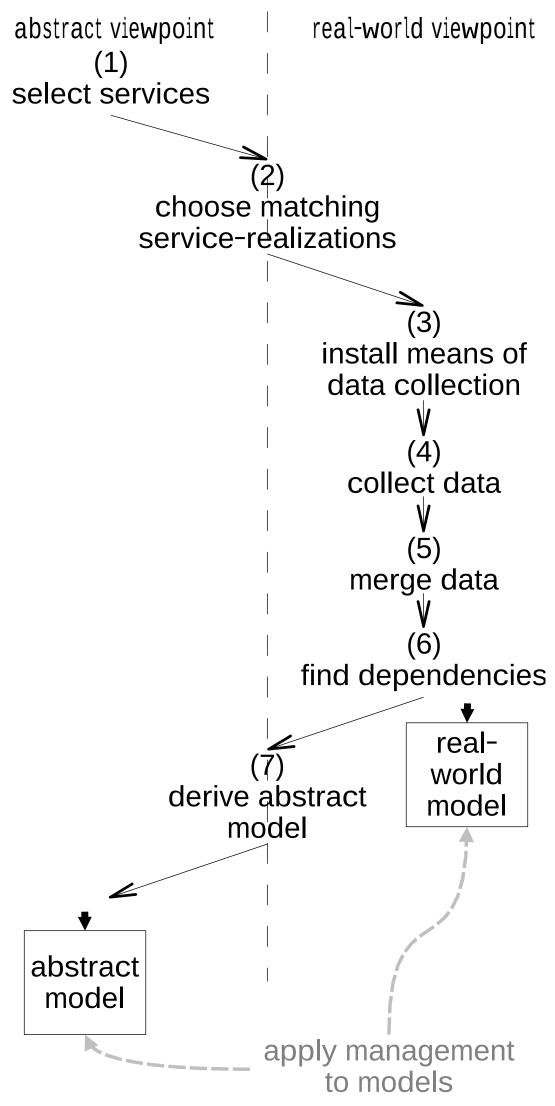

Figure 3: Steps of the Modelling Process

1. First, all services and components that should be part of the model must be selected. Of course this depends on the management tasks the model will be used for.
2. For each abstract service the matching implementations in the real world must be chosen. These may be a number of different software implementations, hardware components, etc. In some cases, as explained in section 3, several of these objects may finally represent only one instance of a service.
3. Now, the necessary means of data collection may be installed per object, using e.g. local management agents or tools commonly available.
4. Then, the real world data may be collected,
5. merged according to the reduced model, and
6. dependencies may be calculated. This step is the actual challenge for the automation using neural technologies, as described in the next section.
7. If necessary for the management task, also an abstract service model can be derived, from the model above, together with the information from step 2.

It is then possible to apply the required management tasks, e.g. as described in section 2, to the models.

The first step is manual, but this is not a real problem, because it is directly driven by the needs of the management. If only models of the real world dependencies are needed, the first two steps may even be omitted; instead, the service implementations must then be selected directly.

The installation of the means of data collection should — together with the process of collection itself — be tied to already used management tools, like management platforms, simple SNMP ([1]) querying software etc. (for a complete overview see [8]). Thus, the effort that must be put into these steps is acceptable.

The most complex task is the final creation of the model (step 6). As it has to deal with even more objects (nodes), it is even more complex than the original task of directly constructing the abstract models. However, with data available from step 5 it is now possible to automate this step. This is described in more detail in the following section.

Figure 4 also shows possible extensions of the process (dotted lines, marked with asterisks) enabling even more applications of the dependency models.

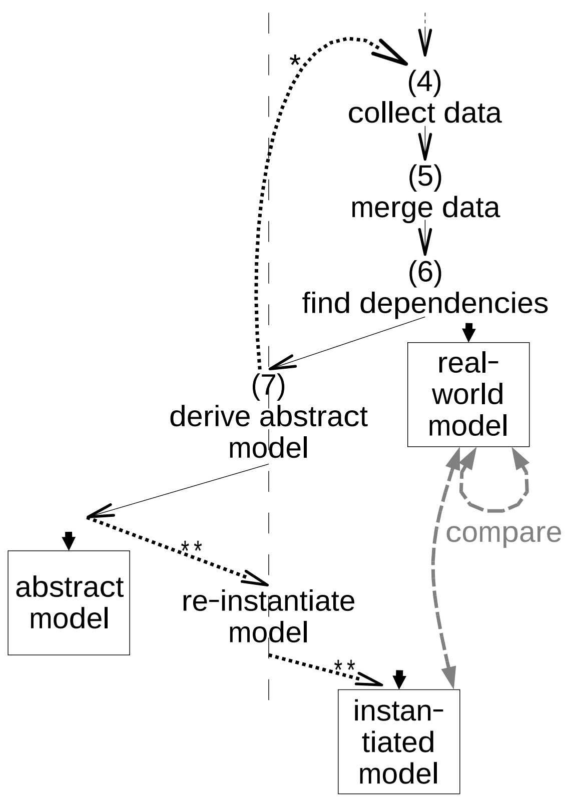

Figure 4: Extension of the Modelling Process

The first types of applications assume the iterative repetition of steps 4 to 6 (*) — respectively 4 to 7, if they work on the abstract models — and take two (or more) models, created at different points in time, as input, e.g. to make out changes that happened in the system.

Such comparisons support *fault prediction*, because changes in system behaviour often reflect errors already present, but simply do not yet effect the usability of services — or at least not to a noticeable extent.

The detected changes may also be used to point out forbidden actions or disallowed use of services. This is helpful especially for *intrusion detection* or to *recognise service misuse.*

Further helpful applications are based on the comparison of the real world models with a re-instantiated abstract model (**), making exceptions and special cases explicitly visible.

Both types of applications take special benefits from the modelling process that are not available in conventional, manually created models.

6 Generating the Models

A basic assumption of the modelling process is that dependencies can be guessed quiet well from the activities of the services, collected in the data of step 5.

A straightforward method to determine the dependencies is to choose data directly expressing this kind of information, like usage entries in log files. Step 5 would then be carried out by extracting the information from the relevant files.

However, a major drawback of this approach is, that log files typically have a proprietary format or sometimes even change between software versions. Even worse, not all applications provide log files containing this information, or its access may be restricted for several other reasons, like e.g. security policies or limited amount of local disk space.

The suggested solution is to concentrate on information which is relatively easy to collect and available for all types of services respectively applications.

Examples for such measurable values allowing to draw conclusions on the services' activities are:

- cpu usage compared to the cpu power available over a certain period of time, or
- communication bandwidth used by the system the service is running on.

Generally speaking, this is information taken from lower layers, like the operating system, middleware or the transport system.

Of course, this information does not show the dependencies explicitly. The fact that two services show activity at the same time does not yet allow to say that they are dependent, but after observing behaviour several times (over a certain period of time), such a conclusion is plausible.

This is where methods from the field of neural networks are able to make use of their advantages, like:

- dealing with uncertain information,
- robustness to noise in the input data

and others also described in [7].

In this case, a neural network is used to determine whether two real world objects have a relationship or not. It is achieved by training the neural network with the data collected from the real environment, for which the results (whether dependencies between the objects exist or not) are known. Examples are needed for both cases.

To achieve good quality, the training set must contain data from at least two or more distinct "service implementation – service user" dependencies as well as pairs of non-related services. Each of them must be observed under various usage conditions and during times of high and low utilisation. Usually, this is the case with data from chosen services in real environments, collected over a longer time period, e.g. a few days including some hours during night and weekend.

During the utilisation of the neural network it may be improved further using reinforcement learning techniques.

Using data from real environments leads to the problem of noisy training data, but with the neural networks ability to generalise these requirements can be met. Furthermore, by this, designing and building a special test field is not necessary — this would even be impossible when hard to set up services have to be modelled.

Figure 5 shows two plots of data collected from two hosts during the same time. The values shown represent the intensity of the hosts' IP-communications with others during time intervals of five seconds.

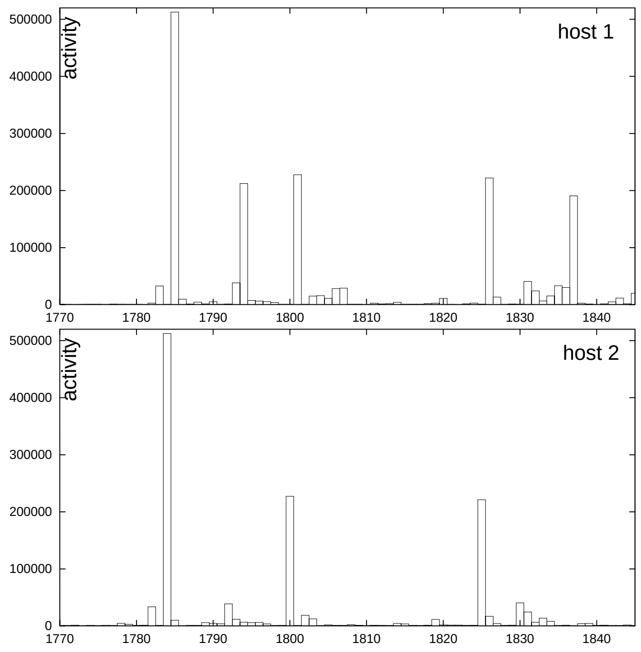

Figure 5: Example plots of network activities of two hosts

Of special interest within the plots are the high spikes. At three time intervals (labelled with the numbers 1785, 1801 and 1826 for the first host, respectively 1784, 1800 and 1825 for the second) both hosts show an activity (of nearly the same intensity) indicating a possible relationship. The plot of host one additionally shows activity at other times (at numbers 1794 and 1837) which is just noise for the investigation of the two hosts' relationship.

In the general case, similar data expressing activity must be selected, as described in the previous section, for each object implementing a service. If several objects have to be merged in the final models the data has to be merged accordingly. This simply can happen by assessing them (assigning factors) and summing the values up.

One problem of this method needs further investigation: In the generation process of real world models of course more than just two objects are involved. To test for all possible relationships of n objects $O(n^2)$ tests are necessary.

On the one hand this argument supports the use of neural networks, as — once trained — they can calculate their tasks faster than traditional correlation analyses. On the other hand it is still a problem for large numbers n.

If only the abstract models are needed finally, it is possible to restrict the modelling process on a very small number of implementations per service. To get complete real world models other restrictions have to be applied. One possibility is to preselect pairs of objects which surely cannot depend on each other. E.g. it is not necessary to test whether two web clients depend on each other. Such exceptions are easy to specify, but significantly reduce the amount of dependencies that have to be investigated.

Another way is to divide the environment that should be modelled into smaller areas, like administrative zones or according to topological aspects. To avoid that these areas must remain absolutely isolated, it is possible to add special objects to each of them representing connections to the outside. This also helps to find the right partitioning: If too many dependencies exist to these objects, it is helpful to add objects to the area. Objects with no (or very few) dependencies within the area are good candidates to remain outside.

7 Conclusion and future work

This article showed that, although dependency models provide a lot of advantages to service management, they are not widely used. The main reason for this is the lack of generally available service models (respectively sufficiently exact dependency descriptions) and the huge effort needed to generate such models manually.

A method was presented that enables the creation of such models for various use cases in a — to a considerable extent — automated way.

There are still unsolved problems and question that have to be investigated further, like to what number of services and applications the method still works well enough. Or what polling intervals are suitable for the data collection. For this, a compromise between reducing time and bandwidth' for the management purposes and the quality of the dependency detection must be found.

To prove and to improve the robustness of the method, especially because it is based on sometimes mistrusted neural networks, the plans for the future also include tests in several real world environments and for various management tasks.

Acknowledgment

The author wishes to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of the paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of Technology, and the Leibniz Supercomputing Center of the Bavarian Academy of Sci-

ences. Its webserver is located at `http://wwwmnmteam.informatik.uni-muenchen.de`.

References

[1] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. RFC 1157: Simple network management protocol (SNMP). RFC, IETF, May 1990.

[2] A. Clemm. *Modellierung und Handhabung von Beziehungen zwischen Managementobjekten im OSI-Netzmanagement.* Dissertation, Ludwig-Maximilians-Universität München, June 1994.

[3] Common Information Model (CIM) Version 2.0. Specification, March 1998.

[4] DMTF Application Management Working Group. Application MOF Specification 2.1. CIM Schema CIM_Application21.mof, Desktop Management Task Force, September 1998.

[5] B. Gruschke. A New Approach for Event Correlation based on Dependency Graphs. In *Proceedings of the 5th Workshop of the OpenView University Association: OVUA'98*, Rennes, France, April 1998.

[6] B. Gruschke. Integrated Event Management: Event Correlation using Dependency Graphs. In A. S. Sethi, editor, *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, October 1998.

[7] Denise W. Gürer, Irfan Khan, and Richard Ogier. An Artifical Intelligence Approch to Network Fault Management. California, USA.

[8] H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application.* Morgan Kaufmann Publishers, 1999.

[9] K. Houck, S. Calo, and A. Finkel. Towards a Practical Alarm Correlation System. In *Integrated Network Management IV*, New York, USA, 1995.

[10] T. Kaiser. *Methodik zur Bestimmung der Verfügbarkeit von verteilten anwendungsorientierten Diensten.* PhD thesis, Technische Universität München, 1999. (To be published).

[11] G. Prem Kumar and P. Venkataram. Network performance management using realistic abductive reasoning model. In A. S. Sethi, Y. Raynaud, and F. Faure-Vincent, editors, *Integrated Network Management IV (ISINM'95).* Chapman & Hall, 1995.

[12] A. Pell, K. Eshghi, J. Moreau, and S. Towers. Managing in a distributed world. In Yves Raynaud and Adarshpal Sethi, editors, *Proceedings of 4th International Symposium on Integrated Network Management.* IFIP, Chapman & Hall, May 1995.